Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 312 (2015) 215–230

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formalisation in Constructive Type Theory of Stoughton's Substitution for the Lambda Calculus

Álvaro Tasistro [1] Ernesto Copello [2] Nora Szasz [3]

Universidad ORT Uruguay,
Montevideo, Uruguay

Abstract

In [25], Alley Stoughton proposed a notion of (simultaneous) substitution for the Lambda calculus as formulated in its original syntax –i.e. with only one sort of symbols (names) for variables– and without identifying α-convertible terms. According to such formulation, the action of substitution on terms is defined by simple structural recursion and an interesting theory arises concerning the connection to α-conversion. In this paper we present a formalisation of Stoughton's work in Constructive Type Theory using the language Agda, which reaches up to the Substitution Lemma for α-conversion. The development has been quite inexpensive e.g. in labour cost, and we are able to formulate some improvements over the original presentation. For instance, our definition of α-conversion is just syntax directed and we prove it to be an equivalence relation in an easy way, whereas in [25] the latter was included as part of the definition and then proven to be equivalent to an only nearly structural definition as corollary of a lengthier development. As a result of this work we are inclined to assert that Stoughton's is the right way to formulate the Lambda calculus in its original, conventional syntax and that it is a formulation amenable to fully formal treatment.

Keywords: Formal Metatheory, Lambda Calculus, Constructive Type Theory

1 Introduction

The Lambda calculus was introduced by Church [5] without a definition of substitution. The complexity of this operation was actually a prime motivation for Curry and Feys to provide the first definition in [7], somewhat as follows:

[1] Email: tasistro@ort.edu.uy
[2] Email: copello@ort.edu.uy
[3] Email: szasz@ort.edu.uy

http://dx.doi.org/10.1016/j.entcs.2015.04.013

$$x[y := P] \quad = \begin{cases} P \ \textit{if}\ x = y \\ x \ \ \textit{if}\ x \neq y \end{cases}$$

$$(MN)[y := P] \ = M[y := P]\ N[y := P]$$

$$(\lambda x.M)[y := P] = \begin{cases} \lambda x.M \ \textit{if}\ y\ \textit{not free in}\ \lambda x.M \\ \lambda x.M[y := P] \ \textit{if}\ y\ \textit{free in}\ \lambda x.M\ \textit{and}\ x\ \textit{not free in}\ P \\ \lambda z.(M[x := z])[y := P] \ \textit{if}\ y\ \textit{free in}\ \lambda x.M\ \textit{and}\ x\ \textit{free in}\ P, \\ \qquad\qquad \textit{where}\ z\ \textit{is the first variable not free in}\ MP. \end{cases}$$

The complexity lies in the last case, i.e. the one requiring to rename the bound variable of the abstraction wherein the substitution is performed. The recursion proceeds, evidently, on the *size* of the term; but, to ascertain that $M[x := z]$ is of a size lesser than that of $\lambda x.M$, a proof has to be given and, since the renaming is effected by the very same operation of substitution that is being defined, such a proof must be simultaneous to the justification of the well-foundedness of the whole definition. This is extremely difficult to formalise in any of the several proof assistants available. Besides, there is the inconvenience that proofs of properties of the substitution operation have to be conducted by induction on the size of terms and have generally three subcases, with two invocations to the induction hypothesis in the subcase considered above. These observations prompt the search for a simpler definition.

As is well known, several of the proposed solutions take the path of modifying the syntax of the language as used above. Such a decision is indeed well motivated, especially if the alternative is to employ for the local or bound names a type of symbol different from the one of the variables: that was, to begin with, Frege's choice in the first fully fledged formal language [11], which featured universal quantification as a binder, and was later made again by at least Gentzen [12], Prawitz [22] and Coquand [6]. Within the field of machine-checked meta-theory, McKinna and Pollack[18] used the approach to develop substantial work in the proof assistant Lego, concerning both the pure Lambda calculus and Pure Type Systems. Now, the method is not without some overhead: there must be one substitution operation for each kind of name and a well-formedness predicate to ensure that bound names do not occur unbound –so that induction on terms becomes in fact induction on this predicate. Another alternative is of course de Bruijn's nameless syntax [8] or its more up-to-date version *locally nameless* syntax [2,4], which uses names for the free or global variables and the indices counting up to the binding abstractor for the occurrences of local parameters. That is to say that locally nameless syntax is a variation of Frege style syntax in which the local parameters are nameless. The overhead in this case is the following: a well-formedness predicate ensures that valid